Simplification Algorithm for Large Polygonal Model in Distributed Environment

Xinting Tang[1], Shixiang Jia[2], and Bo Li[3]

[1] Department of Computer Science and Technology, Ludong University, 264025 Yantai, P.R. China
tangxinting@gmail.com
[2] Department of Computer Science and Technology, Ludong University, 264025 Yantai, P.R. China
jiashixiang@gmail.com
[3] Department of Computer Science and Technology, Ludong University, 264025 Yantai, P.R. China
boli@163.com

Abstract. Polygonal models have grown rapidly in complexity over recent years, yet most conventional simplification algorithms were designed to handle modest size datasets of a few tens of thousands of triangles. We present a parallel simplification method for large polygonal models. Our algorithm will partition the original model firstly, send each portion to a slave processor, simplify them concurrently, and merge them together lastly. We give an efficient method to deal with the problem of partition border and portion merging. With parallel implementation, the algorithm can handle extremely large data set, and speed up the execution time. Experiment shows that our algorithm can produce approximations of high quality.

Keywords: Polygonal model, simplification, parallel, distributed environment.

1 Introduction

Polygonal model simplification has been a hot topic of research over the recent years. It is the process of reducing the number of polygons in a polygonal model while preserving the shape or appearance of the original model. A number of model simplification algorithms have been proposed [1], each with their different strengths and weaknesses in terms of execution time and approximation quality.

An algorithm based upon energy function was put forward by Hoppe [2]. The function synthetically examines the distance from the vertex set X of the original model to the simplified model and the number of the vertices of the simplified model. In order to make the function always find a minimum, a spring item is added. In this algorithm, three basic operations of edge collapse, edge split and edge swap are defined. The algorithm chooses one operation at a time. If the operation can reduce the value of the function, the algorithms adapt it, otherwise another operation will be tried. The new position after collapsing is obtained by an optimized method. This algorithm can produce approximations with high quality, but it is very slow. In

D.-S. Huang, L. Heutte, and M. Loog (Eds.): ICIC 2007, LNCS 4681, pp. 960–969, 2007.

Garland's algorithm, the collapse cost is defined as the sum of squares of the distances from the new point to the triangles adjacent to the two points of the collapse edge in the original model. The position of the new point is obtained by trying to find the point which can minimize the collapse cost [3]. To make the calculation of the collapse cost simple, this algorithm maintains an error matrix for every vertex. When the model gets larger, the consumption of memory can be very huge. There is considerable literature on surface simplification using error bounds. Cohen and Varsheny [4] have used envelopes to preserve the model topology and obtain tight error bounds for a simple simplification. An efficient function of collapse cost has been presented in [5], which is based on the length of contracting edge and the sum of the dihedral angles.

As is common in most areas of computing, improvements in processor speed and memory capacity have served merely to promote the production of increasingly larger datasets, and a number of methods, particularly for out-of-core visualization, have been proposed for coping with models that are too large to fit in main memory, e.g. [6,7]. Following this trend, some of the more recent simplification algorithms have been designed to be memory efficient, and typically handle models with as many as several million triangles. In the last few years, however, there has been an explosion in model size, in part due to improvements in resolution and accuracy of data acquisition devices. These enormous datasets pose great challenges not only for mesh processing tools such as rendering, editing, compression, and surface analysis, but paradoxically also for simplification methods that seek to alleviate these problems. In addition to their large memory consumption, these algorithms also suffer from insufficient simplification speed to be practically useful for simplifying very large meshes.

So, we present a parallel reduction algorithm for these very large models. With parallel implementation, our algorithm can handle extremely large data set, and speed up the execution time. The rest of the paper is organized as follows. We first introduce the basic sequentially simplification algorithm in Section2. Section 3 describes our parallel algorithm in detail. The implementation is discussed in Section 4. Section 5 presents a discussion of results and performance analysis.

2 Basic Simplification Algorithm

We start by giving the basic sequentially simplification algorithm. In Our algorithm, we use half-edge collapse as the atomic decimation operator (see Fig. 1). This operator does not introduce new vertex position but rather sub-samples the original mesh. We prefer half-edge collapse since they make progressive transmission more efficient (no intermediate vertex coordinates) and enable the construction of nested hierarchies that can facilitate further applications.

According to the characteristics of human vision system, observers are mainly sensitive with three attributes of the model: size, orientation and contrast. According to the first attribute, the length of the edge should be considered when calculating its' collapse cost. With the last two attributes, the dihedral angles between the related triangles are also important guidance. Our cost function will focus on the edge length and the sum of the dihedral angles.

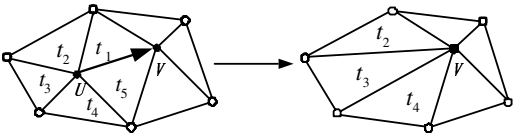


Fig. 1. Half-edge collapse. The (*u, v*) edge is contracted into point *v*. The t_1 and t_5 triangles become degenerate and are removed.

We give every triangle a weight when calculating the dihedral angle. For edge(*u*,*v*) in Fig. 2, when calculating the dihedral angle between t_1 and the other triangles, we think the one between t_1 and t_2 is most important , so the weight of t_2 to t_1 should be largest, and. While, when we calculate the dihedral angle between t_5 and the other triangles, the weights of t_4, t_3, t_2 and t_1 should decrease clockwise.

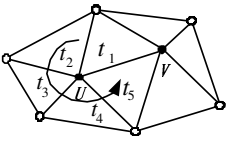


Fig. 2. The calculation of the weight. The weights of t_2, t_3, t_4 and t_5 decreases counterclockwise.

We define that *S* is the set of triangles that are adjacent to vertex *u*, the number of the triangles in it is *n* and s_i (*i*=1, 2 , , ,n) indicates the *i*th triangle. *B* is the set of triangles that are adjacent to both *u* and *v*, the number of the triangles in it is *m*. We define the weight of s_i to b_j as follows:

$$W(s_i, b_j) = n/(n + D(s_i, b_j)) \tag{1}$$

where $D(s_i, b_j)$ in formula 1 denotes the number of triangles between s_i and b_j. In Fig. 2, if b_j is t_1, $D(s_i, t_1)$ denotes the number of triangles which will be visited when traversing counterclockwise from t_1 to s_i. For example, $D(t_2, t_1)$=1, $D(t_4, t_1)$=3. If b_j is t_5, $D(s_i, t_5)$ denotes the number of triangles which will be visited when traversing clockwise from t_5 to s_i.

Define f_i (*i*=1, 2,..., n) indicates the unit normal vector of the *i*th triangle of *S*, and e_j (*j*=1, 2, , , m) indicates the unit normal vector of the *j*th triangle of *B*. We define the collapse cost of edge (*u*, *v*) is:

$$Cost(u,v) = \| u - v \| \times (\sum_{j=1}^{m} \sum_{i=1}^{n} [(1 - (e_j \bullet f_i)) \times W(s_i, b_j)]) \tag{2}$$

where $\|u\text{-}v\|$ in formula 2 indicates the length of edge*(u,v)*.

$$e_j \bullet f_i = | e_j | \times | f_i | \times \cos\theta = \cos\theta \tag{3}$$

We use $e_j \cdot f_i$ in formula 3 to compare the value of the dihedral angleθ, so we can avoid the calculation of arccosine.

3 Parallel Simplification Algorithm

3.1 Overview

The approach for our parallelization is to partition the original high resolution mesh on a master processor and then to send each mesh partition to a slave processor. Each slave processor simplifies its associated mesh subset concurrently. Once this task is complete, each slave returns its simplified result to the master processor which now merges them together to create a single model for the entire mesh (see Fig.3).

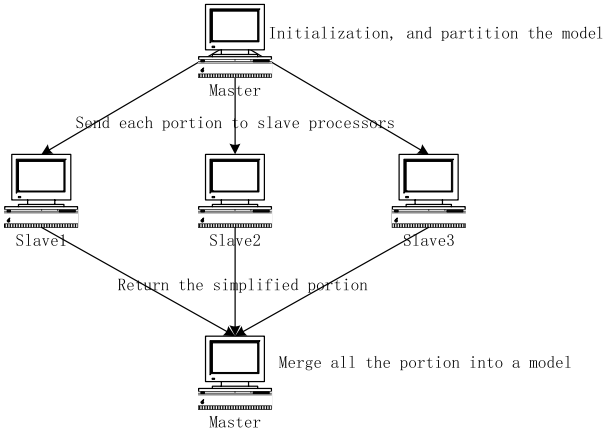


Fig. 3. The processing flow of parallel simplification on four PCs

3.2 Data Partition

The first step in parallelizing simplification is to partition the data. It is an important issue in parallelizing an algorithm. Smaller simplification tasks are created by partitioning the data, and then executing in parallel or serially on a system that is not able to fit the entire model into its core memory. How the data is partitioned greatly impacts the performance of the algorithm and the quality of the simplification. The surface can be subdivided in several ways. The easiest approach is to spatially subdivide the model and assign a portion to each processor. Another approach is to base the partition size on the number of polygons; that is, each processor gets the same number of polygons to simplify. The drawback of these two approaches is that they do not account for the complexity of the underlying surface. We can also initially analyze the surface, and then assign each task a portion of the model based on the complexity of the surface; surface area is part of the surface complexity metric.

In our algorithm, we use a simple greedy method. The mesh is partitioned by accumulating vertices and faces in subsets when traveling through the mesh. A starting vertex is chosen and marked. The accumulation process is performed by selecting the neighbors of the starting vertex, then the neighbors of the neighbors and so on, until the subset has reached the required number of vertices. Then, other subsets are created

the same way until the p-way partition is complete (e.g. each vertex is part of one subset). In the general case, such a p-way partition is built from p partial Breadth-First-Search traversals of the graph. The algorithm terminates when all vertices have been visited. This simple greedy heuristic will yield acceptable partitions in much less time than more complicated methods. Furthermore, the algorithm can be made probabilistic if necessary. It suffices to initialize it with a random vertex seed to generate different partitions for a same input graph. Fig.4 shows one 5-way partition example of this algorithm on a 3D mesh representing a cow.

The subsets being built may get blocked in the process before they reach full size. Then, two versions of the algorithm are possible: allow subset size imbalance or subset multi-connectivity. The former produces uneven subset sizes (workload on processors) and the latter produces bigger edge-cuts (more communication between processors). We chose the latter for better load balancing.

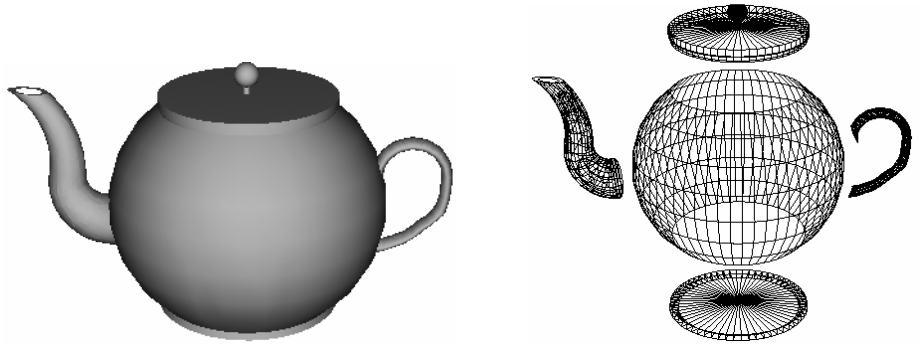


Fig. 4. One 5-way partition of teapot mesh, which has 21,459 polygons

3.3 Partition Border

How to handle the border points is an important problem for parallel reduction algorithm. There are two kinds of border points, one is the border of original model, and we call it original border. The other kind is inner border, which is generated by partitioning. The human eye is sensitive to the border change, so we should try to avoid changing the original border. As to the inner border points, they play an important role in merging the partitions. First of all, we should recognize these border points, and mark them. In the following process of simplification, they are treated specially.

3.3.1 Marking the Border Points

Since the partitioning process traverses the whole mesh, the algorithm can recognize all the border points in the traversing process. In the traversing, we will give every point an only ID, which is helpful to merging. The original border can be easily recognized by the characteristic of its adjacent edges, which is mentioned in many sequentially simplification algorithms. The inner border points can also be found easily. After a vertex is added into a partition, we will select all of its neighbors, and add them into the same partition. If the number of the vertices has already reached the threshold, we won't add this vertex' neighbors, and then we know this vertex is a border point for this partition.

3.3.2 Treatment of the Border Points

As mentioned above, the original border points can't be moved, otherwise the model's border will be changed. For example, v is a border point, and u is a inner point adjacent to it. In order to keep the border unchanged, the vertex v can't be collapsed into u. The inner border points also can't be collapsed, because it will be used when merging the simplified results. If the inner border points are collapsed, we will have to track its path, and this will make merging job become more complex.

Though all the border points can't be shifted, the adjacent inner points can be collapsed into them if necessary (see Fig.5). We have achieved this by adjusting the collapse cost of half edge. When computing the collapse cost of half-edge(u,v), we just calculate it according the cost function. While, we will give a very large value for the collapse cost of half-edge(v,u).

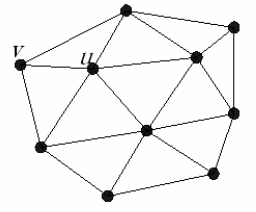


Fig. 5. The border point v can't be shifted to u, but u can be collapsed into v

3.3.3 Merging the Reduced Subset

When partitioning the original model, the inner border points are added at least twice to different portion. According to the treatment to inner border points described above, these points are left unchanged in the simplification process. So we will merge the simplified portion together according the inner border points.

4 Implementation

The reducing process is a master/slave configuration although the amount of extra work the master performs is minimal. The following is the basic outline of our implementation:

```
ParallelReduction(Mesh M, PartitionSize p)
  if (ProcID == 0) //Master
     (M1, ..., Mp) = Partition(M, p)
     for i=1..p
     send Mi to Proci
     for i=1..p
     receive PMi from Proci
     merge PMi into PM
     return PM
  else //Slave section
     receive MProcID from Proc0
     PMProcID = Simplify(MProcID)
```

The code was written in C, using an MPI package for communication. The master processor partitions the mesh into p subsets. The partitioner will return a size |V| integer array. Each array cell corresponds to a mesh vertex and contains a subset ID [1..p] indicating the processor to which the vertex is assigned. The vertex data structure is as follows:

```
struct  vertex {
        double  coordinate[3];
        double  normal[3];
        unsigned int  *vertex_list;
        unsigned int  *face_list;
        unsigned int  vert_num;
        unsigned int  face_num;
}
```

The next step is to send that partition array to the slave processors. Then, all processors read the same mesh file into a Mesh object exactly as in the sequential version. With the partition array at hand, the slave processors build a working mesh structure of edges and faces which are either part of their partition subset or adjacent to it. Next, the slave processors begin to simplify the sub-mesh concurrently. After simplification, the slave processors must transmit their sub-results to the master processor for merging.

5 Results and Discussion

To evaluate the quality and performance of our implementation, we performed a series of tests on a non dedicated cluster of 2,4,8 Pentium IV 2.4GHz PCs, each with 256MB ram and connected by a 100 Mb/s Ethernet. P0 was the master processor. We ran these tests when most of the machines were idle. The performance results are shown in table1. The data is the execution time of reducing the model to the size of 10% of original model. For each input model and number of processors, we performed five test runs, the number in table1 are averaged values.

The acceleration *a* is computed as follows:

$$a = \frac{Time_1}{Time_n} \tag{4}$$

where n=1..8 is a number of processors used and $Time_n$ is the time obtained if n processors are used, and Time1 is the time of sequential algorithm. Fig. 6 shows the acceleration of Bunny model and Dragon model. As expected, speedups are slightly lower for smaller data sets and it should be improved for larger data sets. But these models are too big to be simplified by a sequential algorithm, so we can't get the acceleration.

To test the quality of approximation produced by our algorithm, we use Merto[8] to measure the error between the simplified model and the original. The mean error for dragon model is summarized in Table 2. The output size in table2 is the number of vertices in simplified model.

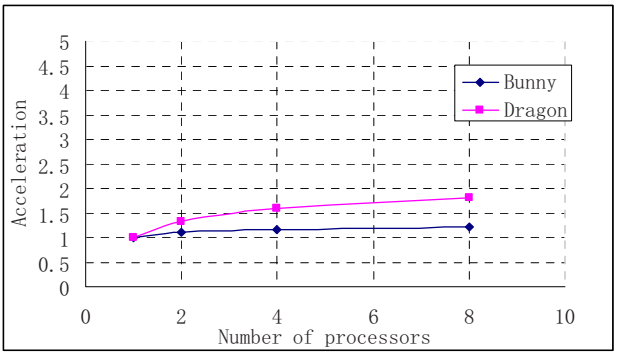


Fig. 6. The acceleration of Bunny model and Dragon model. As expected, speedups are slightly lower for smaller data sets and it should be improved for larger data sets.

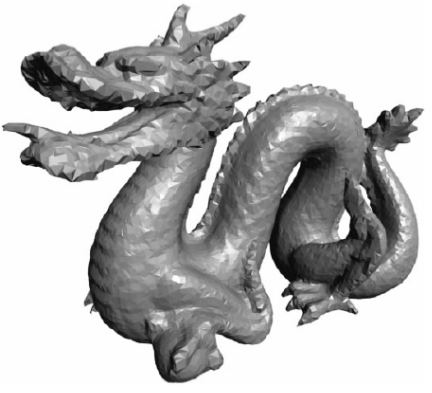
Table 1. Execution time (in second) of various models, with different numbers of processors

Model	Triangles	Processors			
		1	2	3	4
Bunny	69,451	12.185	10.987	10.417	10.016
Dragon	871,306	132.476	101.019	83.715	73.369
Buddha	1,087,474	overflow	95.367	78.527	69.068

(a) Original Buddha, 1,087,716 triangles (b) 60,000triangles

Fig. 7. Some pictures of the models before and after reduction

(c) Original Dragon, 871,306 triangles



(d) 20,000triangles

(e) Original Bunny, 69,451 triangles

(f) 5,000triangles

Fig. 7. *(continued)*

Table 2. Approximation error of Dragon model, with diffierent output size and diffierent numbers of processors

Output Size	Processors			
	1	2	3	4
5,000	0.341	0.352	0.345	0.348
10,000	0.185	0.188	0.187	0.187
20,000	0.132	0.132	0.133	0.130

For the larger models, we can't use Metro to measure the error. We also show some pictures of the models before and after reduction in Fig. 7. The experiments show that the number of processors doesn't significantly affect the quality of the simplified model. Naturally, the ratio between the size of the input and the output model is a key factor to the size of the error.

In this paper, we present a new parallel reduction algorithm for massive models. We have succeeded in simplifying some very large polygonal models in parallel which can't be simplified by a sequential algorithm. In addition, to simplifying very large models, we have also achieved good speed by doing the simplification in parallel.

References

1. Heckbert, P. S., Garland, M.: Survey of Polygonal Surfaces Simplification Algorithms. Technical report, CS Department, Carnegie Mellon (1997) 1-4
2. Hugues, H., Derose, T., Duchamp, T., McDonald, J., Stuetzlet, W.: Mesh Optimization. In: Proceedings of SIGGRAPH '93, (1993)19-26
3. Graland, M., Heckbert, P. S.: Surface Simplification using Quadric Error Metrics. In: Proceedings of ACM SIGGRAPH '97 (1997) 209-216
4. Cohen, J., Varshney, A., Manocha, D., Turk, G.: Simplification Envelopes. In: Proceedings of ACM SIGGRAPH '96 (1996)119-128
5. Jia, S. X., Song, L. H., Zhang, L. F.: Fast Simplification Algorithm for 3D mesh Based on Edge Collapse. In: The First International Conference on Computer Science & Education (2006) 205-208
6. Chiang,Y. J., Silva,C. T., Schroeder, W. J.: Interactive Out-of-Core Isosurface Extraction. In: Proceedings of IEEE Visualization '98 Proceedings (1998)167–174
7. Lindstrom, P.: Out-of-Core Simplification of Large Polygonal Models. In: Computer Graphics(Proceedings of SIGGRAPH 2000), 34 New York (2000) 259-262
8. Cignoni, P., Rocchini, C., Scopigno, R.: Metro: Measuring Error on Simplified Surfaces. Computer Graphics Forum 17 (1998) 167–174